when they hear one of the other hunt group members make a response $Response(j, i, k)$. Thus the value of $QTail$ stays synchronized across hunt group members via the fact that they listen to each other. Similarly, every member r of the hunt group has a dedicated data bus to receive packets. When a data packet finishes on the data bus for r, r increments $QHead(r)$. If $QTail(r) \neq QHead(r)$, r sends $Announce(r, QHead(r))$ on the control bus and thus the input port with that queue position will connect to output port r. This is important becaus while requests are to the group, packets are served by specific ports in the group. Similarly, others in the hunt group will hear r's announcement and increment their own copy of $QHead$ to keep $QHead$ synchronized as is $QTail$.

2. **Knockout Implementation:** There are dependencies between the knockout trees. The simplest implementation passes all the losers from the Position $j-1$ tree to the Position j tree. This would take $k \log N$ gate delays, because each tree takes $\log N$ gate delays. Find a way to pipeline this process such that Tree j begins to work on each batch of losers as they are determined by Tree $j-1$ as opposed to waiting for all losers to be determined. Draw your implementation using 2 by 2 concentrators as your building block and estimate the worst-case delay in concentrator delays.

   Solution: The solution is shown in Figure 7. Essentially, the trees are shown horizontally with the first tree selecting the overall winner, the second selecting the runner-up, and so on until k winners are chose. The key idea is that one does not have to wait till the first tree has finished to start the second tree; as soon as losers are chosen in the first level of the Stage 1 tournament (leftmost tree in the figure) the first four losers are immediately ferried across to start a fresh tournament in Stage 2 starting at time 2 in Level 2. This keeps happening; as soon as a loser is determined in any level it is moved across to the right to the next stage (if one exists) where it either joins a tournament at the next time slot (if the total number of contenders is even) or waits in a delay slot for a "next round match".

   The delays are roughly $O(log_2 N + k)$, or more precisely $\log_2 N + k + 2$ which is a big saving over the naive implementation.

3. **PIM unfairness:** In the Knockout example, using just one tree can lead to unfairness: a collection of locally fair decisions can lead to global unfairness. Surprisingly PIM can lead to the some form of unfairness as well (but not to persistent starvation). Consider a two by two switch, where Input 1 has unlimited traffic to outputs 1 and 2, and input 2 has unlimited traffic to Output 1.

   - Show that, on average, Input 1 will get two grants from Outputs 2 and 1 for half the cell slots and one grant (for Output 2 only) for the remaining cell slots. What fraction of Output 2's link does Input 1 receive.

Solution: This is from Nick Mckeown's iSLIP paper. Because of the random and independent selection by the arbiters, Output 1 will grant half the time to Input 1 and half the time to Input 2. On the other hand, since Output 2 only gets requests from Input 1, it grants to Input 1 in all cell times. Thus Input 1 gets 2 grants (from both Outputs 1 and 2) in half the cell slots, and 1 grant (for Output 2) in the remaining half (on average). Thus Input 1 should accept output 1 only a quarter of the time (being fair to 2 requests half the time), and thus Input 1's traffic receives a 1/4 share of Output 1 and a 3/4 share of Output 2. This is somewhat unfair because the MaxMin share of Output 1 should 1/2.

- Infer based on the above the fraction of Output 1's bandwidth that Input 1 receives on average versus Input 2. Is this fair?

4. **Motivating the iSLIP pointer increment rule:** The following is one unfairness scenario if pointers in iSLIP are incremented incorrectly. For example, suppose in Figure **??** suppose that input port A always has traffic to output ports 1, 2, and 3, whose grant pointers are initialized to A. Suppose also that input ports B and C also always have traffic to 2. Thus initially A, B, and C all grant to 1 who chooses A. In the second iteration, since input port B has traffic to 2, 2 and B are matched.

   - Suppose 2 increments its grant pointer to C based on this second iteration match. Between which port pairs can traffic be continually starved if this scenario persists?

     Solution: Traffic can be starved from A to 2 because in the next iteration, A has a grant only from 1, and thus will be accepted. Similarly, 2 will grant to C and this will also be accepted, after which 2 will increment is grant pointer to A again. In the next iteration, A gets a grant from 2 and 1, and the cycle repeats, locking out traffic from A to 2.

   - How does iSLIP prevent this scenario?

     Solution: Clearly, if B does not increment in the second iteration, B will grant in the next round to A again, and this time A will accept because its accept pointer has gone to 2. In general, nobody can be locked out because eventually the grant pointer will not be incremented past an input port unless there is first round accept, in which case progress has been made.

5. **ESLIP:** Answer the following questions about ESLIP.

   - Describe a scenario where a multicast cell does not finish its fanout in one cycle despite the use of a shared grant pointer and the fact that multicast has priority over unicast in alternate time slots.
   - Why is there no need for a shared multicast accept pointer?